Real Time Operating Systems (RTOS)

=

Echtzeit-Betriebssysteme

Eine Einführung

Klaus Kusche, Okt. 2011

Ziele des Vortrags

- *Überblick* über das Thema
- *Praktisches Verständnis* von
  - Anforderungen
  - Problembereichen
  - Aufbau und Bestandteilen
  - Alternativen Lösungsansätzen
- **Nicht** Ziel:
  - Technische / wissenschaftliche Details
  - Interne Funktion der Realisierungen

Vorkenntnisse

- Allgemeine *Betriebssystem-Kenntnisse*

  Verständnis „normaler“ Betriebssysteme (Windows, Linux, …):

  - *Prozesse* und Scheduling
  - (Virtuelle) *Speicherverwaltung* und Swapping
  - *Interprozess-Kommunikation* inkl. Locks / Mutexes
- Verständnis von *I/O, Interrupts und Timern* in Hardware und Software (Treiber, …)

Einsatzbereiche von RTOS

Überall, wo vorhersagbares / deterministisches zeitliches Systemverhalten gefordert ist!

- *Steuerungen*, Messwert-Erfassung, Embedded Systems, Sicherheitssysteme, ...
- *Audio- / Video*-Verarbeitung (z.B. TV-Streaming)
- *Telekommunikation* (z.B. Vermittlungsrechner)
- „Schüttelkinos“ (z.B. Piloten-Trainingssysteme)
- ...

Die zentrale Anforderung

- **Garantierte Reaktionszeit („Latenz“)**
  - zwischen *Hardware-Ereignis = Interrupt*
    (Eingangsdaten, Timer-Ablauf, Störung, …)
  - und *Reaktion*
    (Berechnung, Ausgangsdaten, Anzeige, …)
- … und zwar nicht im Mittel, sondern im

„Worst Case“!

- Manchmal auch:

  Limitierter *Jitter* (= Streuung, Schwankung der Reaktionszeit)

Harte und weiche Echtzeit

- **Hartes** Echtzeit-Betriebssystem:
  Erforderlich, wenn *Zeitüberschreitung fatal* ist:

  Die Reaktionszeiten sind
  berechen- und garantierbar

- **Weiches** Echtzeit-Betriebssystem:
  Nur, wenn *Zeitüberschreitung tolerierbar* ist:
  - Alle Features eines RTOS sind vorhanden
  - Aber nur *„Best Effort“*, *keine Garantien*

  ==> Zeit hält *„praktisch immer“* (> 99.99... %)

Beispiele Reaktionszeiten

- Gebäude-Steuerung: *Mehrere Sekunden*
- Grafisches Benutzer-Interface: *~ 50 - 100 ms*
- Video, Schüttelkino: *1/30 - 1/60 Sek. (60 FPS)*
- Audio, Telefonvermittlung: *1 - 10 ms*
- Steuerungen, Mess-Systeme:

  Bis hinunter zu wenigen Mikro-Sekunden!

Schnellere Hardware statt RTOS ???

- … *hilft nicht* gegen konzeptionelle Defizite „normaler“ Betriebssysteme!

  (es fehlen zeitlich Zehnerpotenzen!)

- … ist bei Echtzeit-typischen Anforderungen oft sogar *langsamer* als „primitive“ CPU’s!

  Beispiele von x86-Performance-Schwächen:

  - Interrupts
  - Taskwechsel
  - Synchronisation / Locks zwischen Cores

RTOS = Schnelles Betriebssystem ???

- Ein Echtzeit-Betriebssystem ist nicht einfach „schneller“, sondern intern *anders aufgebaut!*
- Bei „normalen“ Benchmarks sind RTOS meist sogar einige Prozent *langsamer* als normale OS!

  Grund: Mehr interner Overhead

  Denn:

Ein RTOS ist nicht
Gesamt-Durchsatz-optimiert!

„Wichtiges“ und „Unwichtiges“...

Steuerungs-Software ist typischerweise *hoch parallel,*
macht Dutzende Dinge „quasi gleichzeitig“

==> *nicht für alle Berechnungen*
gilt dieselbe Reaktionszeit-Anforderung!

Sondern:

Reaktionszeit je nach Wichtigkeit der Aufgabe

==> Jeder Task / Prozess / Thread hat eine

Priorität

==> Ein RTOS ist per Definition *nicht „fair“!*

Beispiele Prioritäten

Von „unwichtig“ nach „wichtig“ (unvollständig):

- Logging, Webinterface
- Lokales Benutzer-Interface
- Nicht-Echtzeit-I/O (Disk, Ethernet, USB)
- Statistische / langfristige Berechnungen
- Timer-Code
- Echtzeit-I/O, regelkritische Berechnungen
- Fatale Fehler

Zentrale Anforderung

... an die *Schnittstelle* eines RT-Betriebssystems
(zusätzlich zu normaler System-Funktionalität
wie I/O, Interprozess-Kommunikation, ...):

- **Festlegen der Prozess-Priorität**
  jedes einzelnen Prozesses
- **Festlegen der Scheduling-Strategie**
  bei gleicher Priorität (Round Robin oder FIFO)

Standardisierte Realtime-Systemschnittstelle
im *POSIX-Standard, Kapitel 4!*

Anforderungen an den Scheduler

Deterministischer, präemptiver Scheduler:

- Strikt nach *Priorität*:

  Höherpriore Tasks verdrängen
  niederpriore Tasks *sofort und vollständig*.

- Nach definierten Regeln bei Tasks
  mit derselben Priorität:

  Round-Robin (reihum timesliced) oder
  FIFO (in Ready-Reihenfolge bis fertig)

- Effizient, mit *konstantem Overhead* („O(1)“)

Weitere externe Anforderungen

- **Memory Locking eines Prozesses im RAM:**
  Notwendig wegen Swap-Verzögerungen:
  Echtzeit-Prozesse (Code & Daten) dürfen *nie swappen!*
- **Binden von Prozessen an CPU-Cores:**
  Notwendig wegen Cache-Effekten:
  Cache Misses können mehr als zehnfach verlangsamen!
- **Hochauflösende Timer** (Mikro-Sekunden):
  Normale PC / Windows-Zeitauflösung ist nur 1/60 Sek.!
- Im Audio/Video-Bereich: Reservierung
  garantierter Disk- und Netzwerk-Durchsätze
  für einzelne Prozesse.

Interne Anforderungen

- In **System & Treibern: Fixe maximale Zeit** für
  - alle internen *Locks und Mutexes*
  - alle *Interrupt-Sperren*
  - alle *Interrupt Service Routinen*
  - alle im RT-Kontext laufenden *Systemfunktionen* (z.B. Speicherverwaltung!)
- Auch **Interrupts** müssen **priorisierbar** sein:
  - Echtzeit-I/O, Timer: *Hoch*
  - Disk, Ethernet, Keyboard, ...: *Niedrig*

Interne Codierung

- RTOS-interne *„Langläufer“* (z.B. Aufräum-Code) werden in *asynchrone Threads* verlagert

  ==> laufen verdrängbar mit niedriger Priorität!
- Auch der *Betriebssystem-Code* selbst, sogar Teile der Interrupt Service Routinen, sind *unterbrechbar und verdrängbar*

  ==> hochpriore Interrupts / Tasks verdrängen niederpriore Tasks auch *innerhalb* des Betriebssystems!

Interne Interrupt-Behandlung

Die Interrupt-Behandlung in RTOS ist zweiteilig:

- **ISR** (Interrupt Service Routine) = *„Top half“*:
  - Läuft im Interrupt-Kontext (Interrupts gesperrt)
  - Setzt die Bottom Half auf „runnable“
  - Macht *kein* I/O ==› „sofort“ fertig
- **DSR** (Deferred Service Routine) = *„Bottom half“*:
  - Läuft als normaler Thread (mit Interrupts offen!), später wenn Zeit ist (je nach Priorität)
  - Macht das eigentliche Hardware-I/O

Weitere RTOS-Features

- **Treiber & Protokolle für Echtzeit-Bussysteme**
  (CAN, Ethercat, Profibus/Profinet, Sercos, ...)
- Spezielle **Flash-Filesysteme** für Onboard-Flash
  (weil CF-Card, SSD, ... sind *nicht* RT-tauglich!)
- Interner **Watchdog**
  (gegen Deadlocks, High-Prio-Hänger, ...)
- Konfigurierbares **Hardware-Powersave**
  (Powersave moderner Prozessoren und I/O's
  ist *fatal für RT ==> Abschalt-Möglichkeit!*)

„Monolithisches“ RTOS

= Echtzeit-Betriebsssystem mit

klassischem „All-In-One“-Aufbau

(z.B. umgebautes „normales“ System)

- *„Großer“ Kernel* (ev. Millionen Lines of Code!) erledigt alle OS-Aufgaben
- *Gemeinsame RT- und nicht-RT-Welt*

  (RT- und nicht-RT-Prozesse *unterscheiden sich nur durch ihre Priorität*)

„Monolithische“ RTOS-Beispiele

Angepasste *universelle Betriebssysteme*:

- **Linux mit Echtzeit-Patch** *(Thomas Gleixner)*

  ==> Nur *„weiche“* Echtzeit!

- *Nicht*, obwohl oft genannt: Windows CE

Dedizierte RTOS:

- **VxWorks, QNX, LynxOS, OS/9**, ...

Anforderungen sicherer Systeme

Oft *formale Anforderungen* (für KFZ, Luftfahrt, Medizin, „Not-Aus-Steuerung“, Kernkraftwerke, ...):

- **Zertifizierung**
  ==> erfordert spezielle *Entwicklungsabläufe*
  ==> erfordert umfangreiche *Tests*
- Formale mathematische **Verifikation**
  der *Korrektheit* und des *Zeitverhaltens*
- **Eingeschränkter Sprachumfang**
  (z.B. Automobilindustrie: *MISRA C*)

Software für sichere Systeme

Konsequenz:

- **Bestandscode** darf **nicht verwendet** werden!
- Gesamter Code muss im **Source** vorliegen!
- Jede Codezeile ist extrem
  teuer und aufwändig!

==> Machbar nur für

wenige 1000 / 10000 Lines of Code!

==> *Schließt große, monolithische RTOS*
und universelle Systeme (Windows, Linux) aus!

RTOS-Probleme auf Industrie-PC's

Problem Treiber:

- *3D-Grafik-Treiber, USB- oder WLAN-Treiber*, ...
  sind extrem komplex, umfassen sehr viel Code
- HW-Hersteller unterstützen
  nur Windows & Linux, keine RTOS
- RTOS-Hersteller sind mit Treibern überfordert

==> *Dedizierte RTOS*
unterstützen moderne x86-PC's sehr schlecht!

==> *„Normales" I/O*
sollte via Windows/Linux gemacht werden!

RTOS mit Mikrokernel

Der **Mikrokernel** *partitioniert* die Hardware
in *mehrere getrennte Prozess-Bereiche*:

- Ein oder mehrere Bereiche enthalten die **Echtzeit-Prozesse**.
- Ein weiterer Prozess-Bereich enthält ein **„normales“ Betriebssystem (Windows / Linux)** *als vom Mikrokernel verwalteten Prozess*.
- *Nicht-Echtzeit-Prozesse* (Benutzer-Interface usw.) laufen *innerhalb des normalen Betriebssystems* (ohne Zutun des Mikrokernels).

Aufbau des Mikrokernels

Der Mikrokernel enthält *nur*:

- **Prozessverwaltung** & Scheduler
- **Interrupt-Verteilung** (Top Half) und Timer
- **Speicherschutz** und -Verwaltung
  (meist nur *statisch*)
- **Interprozess-Kommunikation**, Locking, ...

... und sonst nichts!

Vorteile von Mikrokernel-RTOS

==> Das *normale Betriebssystem* macht die gesamten *nicht-Echtzeit-I/O's* (Disk, Netz, USB, ...) incl. *Grafik*.

==> Die Echtzeit-Prozesse leiten ihre nicht-Echtzeit-I/O's *via Mikrokernel an das normale Betriebssystem weiter*.

==> Der Mikrokernel und die Echtzeit-Prozesse enthalten *keine Standard-I/O- oder Grafik-Treiber*, *kein Filesystem, keine Netzwerk-Protokolle, ...*

Der Kernel ist daher wirklich „Mikro".

Nur der Mikrokernel und die Echtzeit-Bereiche müssen zertifiziert / verifiziert werden!

Nachteile von Mikrokernel-RTOS

- **2 getrennte Welten**
  für RT- und nicht-RT-Programme:
  - Andere *Entwicklungsumgebungen*
  - Andere *System-Schnittstellen und Libraries*
  - Meist *keine direkte Kommunikation*
    (kein Shared Memory)
- Mikrokernel-Lösungen sind meist **langsamer**
  (mehr Prozess-Wechsel, mehr Kommunikation, ...).

Mikrokernel-RTOS-Beispiele

- **RTAI, Xenomai, RTLinux:**

  Open Source und/oder kommerziell,
  mit *Linux* als normalem System

- **PikeOS:**

  Kommerziell, u.a. auch mit *Linux*
  oder eigenen I/O-Subsystemen
  (Windows in Planung)

- **RTX:**

  Kommerziell, mit *Windows*